## An introduction to automata design with Ti*k*Z's automata library

Igor Borja

**Abstract**

This article is a quick introduction to Ti*k*Z's *automata* library, used for the design and typesetting of finite automata in LaTeX. It also explores the use of TeX loops and conditionals to automate the generation of images that follow noticeable patterns. Ti*k*Z itself is a package used for generating a variety of figures — from geometry configurations to graphs and automata — allowing for more control over image editing and quality. Although the package is very versatile, its uses for designing automata will be the primary topic of this article.

### 1 Introduction and basic syntax

Finite automata, also called finite state machines, are a basic concept in computer science for modeling computation. Wikipedia (`en.wikipedia.org/wiki/Finite-state_machine`) provides an introduction to the topic.

In this article, all the code to typeset an automaton will be contained inside a `tikzpicture` environment [3]. After starting the environment, you can pass optional arguments, separated by a comma, such as *node distance* and *arrow style*. A reminder: the node distance (which we'll see below) must be a dimension (`cm`, `em`, `pt`, etc.).

#### 1.1 Nodes

You can declare a node of an automaton via the following syntax:

`\node[state,` ⟨*state modifiers*⟩`,`
⟨*position modifiers*⟩`] (`⟨*id*⟩`) {`⟨*name*⟩`};`

Note that a node declaration *should always end in a semicolon*. Let's analyse all of these parameters:

1. Every state node must begin with the word `state`.
2. State modifiers are mostly used to indicate that node is the `initial` node or an `accepting` node, and thus are often not needed.
3. Position modifiers are used to place a node relative to another (*already declared*) node. Some common modifiers are `right=of` ⟨*id*⟩, `left=of` ⟨*id*⟩, `below=of` ⟨*id*⟩ and `above=of` ⟨*id*⟩. Here, ⟨*id*⟩ is the id of the node relative to which the positioning is carried out.

   Also, it's possible to give dimensions via the `xshift` and `yshift` parameters to achieve manual control over the position after the relative placement.
4. The ⟨*id*⟩ is the *unique* identifier that will be used to refer to that node later.
5. The ⟨*name*⟩ is the text that will appear in the automaton, inside the circle that represents that node. It does not need to be plain text; it's also possible to use a math expression (enclosed by single `$` signs).

A minimal working example:

```
\documentclass{standalone}
\usepackage{tikz}
\usetikzlibrary{automata}
\usetikzlibrary{positioning, arrows}
\begin{tikzpicture}
    [->, node distance = 2cm]
    \node[state] (p) {$p = 1$};
    \node[state, accepting, right=of p,
        yshift= -2cm] (q) {$q = 2$};
\end{tikzpicture}
```

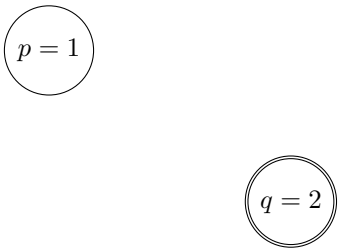


**Figure 1**: Two nodes

The use of ⟨*direction*⟩ `of=`⟨*id*⟩, although correct, is marked as deprecated in PGF/Ti*k*Z source code [4].

#### 1.2 Edges

You can declare an edge between two nodes with the following syntax

`(`⟨*id-head*⟩`) edge[`⟨*options*⟩`]`
`node[`⟨*options*⟩`]{`⟨*value*⟩`} (`⟨*id-tail*⟩`);`

Every sequence of consecutive edge declarations must be preceded by a `\draw` command. Also, a semicolon is used to indicate the last edge declaration of a sequence; any that come after that and before another `\draw` command will be ignored. Therefore (provided `n1, n2, n3` have all been declared), this is correct:

```
\draw
(n1) edge node{text} (n2)
(n2) edge node{more text} (n3)
(n2) edge node{more text} (n1);
\node[state, above=of n1] (n4) {text};
\draw
(n3) edge node{more text} (n4);
```

While this is (for both reasons mentioned above) not correct:


doi.org/10.47397/tb/44-1/tb136prado-automata

```
\draw
(n1) edge node{text} (n2)
(n2) edge node{more text} (n3)
(n2) edge node{more text} (n1)
\node[state, above=of n1] (n4) {text};
(n3) edge node{more text} (n4);
```

The edges are treated as directed — to get the visual effect of undirected edges, remove the arrow in the environment options.

### 1.2.1 Analysing the different components of an edge command

⟨*id-head*⟩ is the identifier of the *head* node — the node from which the edge is originated.

⟨*id-tail*⟩ is the identifier of the *tail* node — the node to which the edge arrives.

The options after the `edge` keyword indicate how the edge should be drawn. The following options are the most common:

- Directions `right`, `left`, `below`, `above`: indicate from where the edge should leave
- `loop`: specifies that the edge should loop and go back to the head node. Using the `loop` option makes Ti*k*Z ignore the tail node's id (which can be left empty).

  Combining the `loop` option with directions indicates where the loop should be rendered: above, to the right, left or below the node.
- `bend`: indicates bends in the edge to a certain direction

The options after the `node` keyword indicate how the text associated with that edge should be drawn. The most common arguments are the four main directions `right`, `left`, `above` and `below`.

Defaults: if no `edge`-positioning options are given, the edge will be drawn as a straight line by default. Also, if no `node` options (i.e., options that determine the positioning of the edge text) are provided, the text will be rendered at the "center" of the edge by default. See both behaviors below:

```
\documentclass{standalone}
\usepackage{tikz}
\usetikzlibrary{automata}
\usetikzlibrary{positioning, arrows}
\begin{tikzpicture}
    [->, node distance = 2cm]
    \node[state] (p) {$p = 1$};
    \node[state, accepting, right=of p,
        yshift=-2cm] (q) {$q = 2$};
    \draw
    (p) edge node{hello} (q)
    (p) edge[loop] ();
\end{tikzpicture}
```

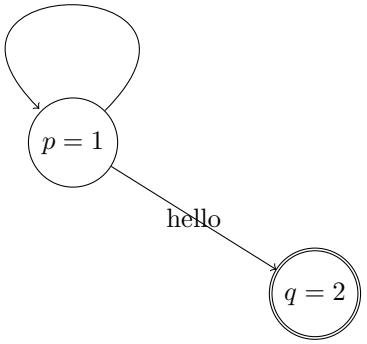


**Figure 2**: Two nodes, a labeled edge and a loop

## 2 Loops and automation by example

Here we show a more complex example.

### 2.1 Context

Consider the language $L$ over the alphabet $\Sigma = \{a, b\}$ that contains all the sequences with at least 2 characters $a$ and at least 1 character $b$. Note that $L$ is the intersection of two regular languages (over the same alphabet $\{a, b\}$): the set of strings with at least 2 characters $a$ and the set of strings with at least 1 character $b$.

Therefore, using the construction detailed in [2], a possible finite automaton that recognizes $L$ is $M = (Q, \{a, b\}, \delta, q_{0,0}, q_{2,1})$, where

$$Q = \left\{ q_{i,j} \,\middle|\, 0 \leq i \leq 2 \wedge 0 \leq j \leq 1 \right\}$$

are the states that can be reached. The transition function $\delta$ works as follows: $\delta(q_{i,j}, a) = q_{\min(2,i+1),j}$ and $\delta(q_{i,j}, b) = q_{i,\min(1,j+1)}$.

In other words, $q_{i,j}$ represents that the string read has (up to that point) $i$ characters $a$ (if $i < 2$) or 2 or more, if $i = 2$. Also, it has $j$ characters $b$ if $j < 1$, else it has 1 or more. Representing it graphically, we get the following state diagram:

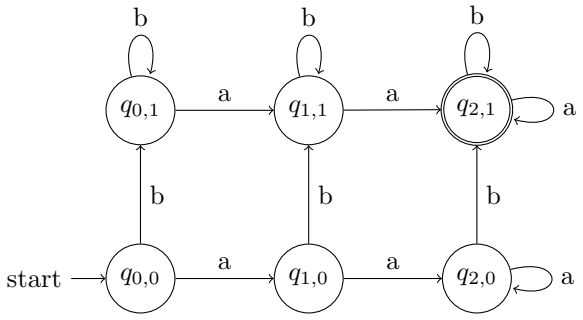


**Figure 3**: State diagram

which we can produce with the following LaTeX code:

```
\begin{tikzpicture}
    [->, node distance = 1.3cm]
    \node[state, initial] (a0b0)
        {$q_{0,0}$};
    \node[state, right=of a0b0] (a1b0)
        {$q_{1,0}$};
    \node[state, right=of a1b0] (a2b0)
        {$q_{2,0}$};
    \node[state, above=of a0b0] (a0b1)
        {$q_{0,1}$};
    \node[state, above=of a1b0] (a1b1)
        {$q_{1,1}$};
    \node[state, accepting, above=of a2b0]
        (a2b1) {$q_{2,1}$};
    %% Horizontal edges in first layer
    \draw
    (a0b0) edge node[above]{a} (a1b0)
    (a1b0) edge node[above]{a} (a2b0)
    (a2b0) edge[loop right]
        node[right]{a} (a2b0);
    %% First set of vertical edges
    \draw
    (a0b0) edge node[right]{b} (a0b1)
    (a1b0) edge node[right]{b} (a1b1)
    (a2b0) edge node[right]{b} (a2b1);
    %% Horizontal edges in second layer
    \draw
    (a0b1) edge node[above]{a} (a1b1)
    (a1b1) edge node[above]{a} (a2b1)
    (a2b1) edge[loop right]
        node[right]{a} (a2b1);
    %% Second set of vertical edges
    \draw
    (a0b1) edge[loop above]
        node[above]{b} (a0b1)
    (a1b1) edge[loop above]
        node[above]{b} (a1b1)
    (a2b1) edge[loop above]
        node[above]{b} (a2b1);
\end{tikzpicture}
```

It is easy to see that this state diagram's grid-like structure generalizes nicely to the family of languages $(L_{m,n})_{m,n\in\mathbb{N}}$, where $L_{m,n}$ is the language of strings with at least $m$ characters $a$ and at least $n$ characters $b$.

However, drawing the state diagram for $L_{m,n}$ quickly becomes too much work, since there will be $(m+1)(n+1)$ nodes and $2(m+1)(n+1)$ edges: even at small values (say, $m = 4$ and $n = 3$) it is still more than 60 lines of code.

### 2.2 Loops

However, due to its very simple structure, rendering automata like this one can be abstracted and automated in a relatively straightforward way, using `foreach` loops, available through the package `pgffor`.

1. For the nodes, it's possible to draw first $q_{0,0}$ with id `a0-b0`, then for each $1 \leq i \leq m$ draw $q_{i,0}$ at the right of $q_{i-1,0}$ and attribute to it the id `a`$\langle i \rangle$`-0` (where $\langle i \rangle$ is a placeholder for the value of $i$). This completes the first row.

   Then, for each $1 \leq j \leq n$ and for each $0 \leq i \leq m$ we draw $q_{i,j}$ above $q_{i,j-1}$ and attribute to it the id `a`$\langle i \rangle$`-b`$\langle j \rangle$. At each iteration it is necessary to check if $i = m$ and $j = n$, in which case we need to add the `accepting` option.

2. For the edges, we have four different cases: standard horizontal edges, standard vertical edges, edges that loop above the node and edges that loop at the right of the node.

   For each $0 \leq i \leq m-1$ and $0 \leq j \leq n$, we build a horizontal edge from $q_{i,j}$ to $q_{i+1,j}$. Then, for each $0 \leq i \leq m$ and $0 \leq j \leq n-1$ we build a vertical edge from $q_{i,j}$ to $q_{i,j+1}$. Finally, we build an edge that loops above $q_{i,n}$ for each $0 \leq i \leq m$ and an edge that loops at the right of $q_{m,j}$ for each $0 \leq j \leq n$.

It is necessary to refer to $i-1$ and $j-1$ several times in this algorithm. However, simply using the TeX code `\i - 1` and `\j - 1` and `a\i - 1b\j - 1` (for ids) in the TeX code won't work: loop indices in `foreach` loops function as standard variables defined with `\def` (just with restrained scope). That means that any reference `\i` will be replaced by the value of `\i` (as a string), so the expression `\i - 1` will not be evaluated. For example, if $i = 5$, `\i - 1` will be replaced by `5 - 1`.

In order to fix that, we use the fixed-point arithmetic package called `fp` and its command for evaluating expressions: `\fpeval`. The implementation below summarizes all that in working LaTeX code, abstracting it all in a command with three arguments called `\gridAutomata`. The first argument is the node distance, the second is $m$ and the third is $n$:

```
\usepackage{ifthen}
\newcommand{\gridAutomata}[3][2cm]
{
\begin{tikzpicture} [
    ->,
    node distance = #1,
]
%% building first row of nodes
\ifthenelse{\equal{#2}{0}}
{
    % if m == 0
    \ifthenelse{\equal{#3}{0}}
    {
        \node[state, initial,
```

```
            initial where=below, accepting]
            (a0-b0) {$q_{0,0}$};
    }{
        \node[state, initial,
            initial where=below]
            (a0-b0) {$q_{0,0}$};
    }
}{
    % if m > 0
    \node[state, initial, initial where=below]
        (a0-b0) {$q_{0,0}$};


    \foreach \i in {1,...,\fpeval{#2 - 1}}
    {
        \node[state,
            right=of a\fpeval{\i - 1}-b0]
            (a\i-b0) {$q_{\i, 0}$};
    }
    \ifthenelse{\equal{#3}{0}}
    {
    % if m > 0 and n == 0
        \node[state,
            right=of a\fpeval{#2 - 1}-b0,
            accepting]
            (a#2-b0) {$q_{#2, 0}$};
    }{
        % if m > 0, n > 0
        \node[state,
            right=of a\fpeval{#2 - 1}-b0]
            (a#2-b0) {$q_{#2, 0}$};
    }
}
%% other rows
\foreach \j in {1,...,#3}
{
    \foreach \i in {0,...,#2}
    {
        \ifthenelse{\equal{\i}{#2}
                    \AND \equal{\j}{#3}}
        {
            \node[state,
                above=of a\i-b\fpeval{\j-1},
                accepting]
                (a\i-b\j) {$q_{\i, \j}$};
        }{
            \node[state,
                above=of a\i-b\fpeval{\j-1}]
                (a\i-b\j) {$q_{\i, \j}$};
        }
    }
}
%% Constructing the edges
%% Loops above
\foreach \i in {0,...,#2}
{
    \draw (a\i-b#3) edge[loop above]
        node[above]{b} (a\i-b#3);
}
%% Rightmost loops
\foreach \j in {0,...,#3}
{
    \draw (a#2-b\j) edge[loop right]
        node[right]{a} (a#2-b\j);
}
%% Horizontal edges
\foreach \i in {0,...,\fpeval{#2 - 1}}
{
    \foreach \j in {0,...,#3}
    {
        \draw (a\i-b\j) edge
        node[above]{a} (a\fpeval{\i+1}-b\j);
    }
}
%% Vertical edges
\foreach \j in {0,...,\fpeval{#3 - 1}}
{
    \foreach \i in {0,...,#2}
    {
        \draw (a\i-b\j) edge
        node[right]{b} (a\i-b\fpeval{\j + 1});
    }
}
\end{tikzpicture}
}
```

Using this command for $m = 4, n = 3$ with a node distance of 1.5 cm yields the result below:

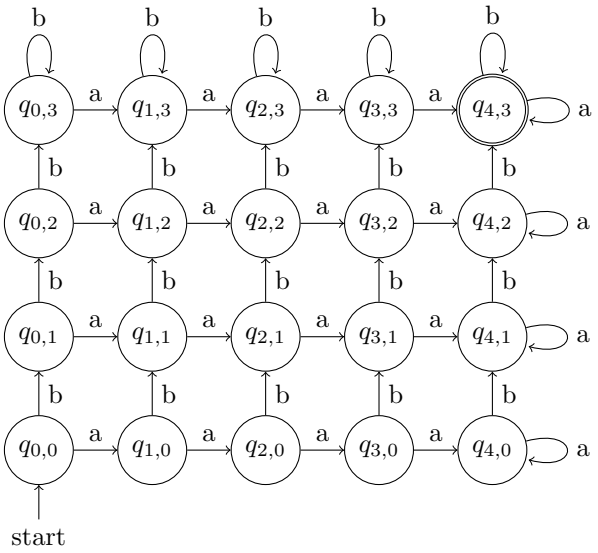


**Figure 4**: Result for $m = 4,\ n = 3$

This command's generality allows for construction of a larger example. The image on the next page uses the values $m = 7, n = 8$ with a node distance set to 1.5 cm.

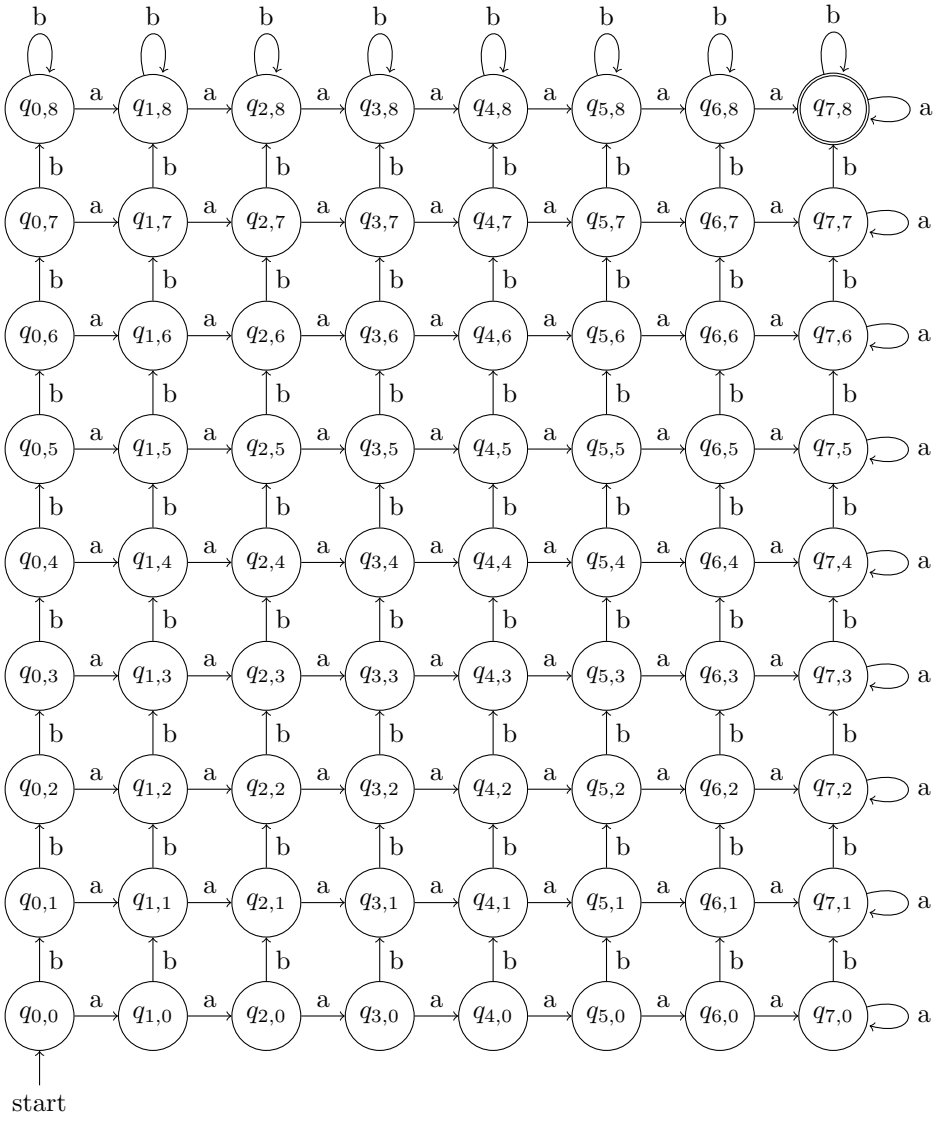


**Figure 5**: Result for $m = 7, n = 8$

## 3 More customization possibilities

### 3.1 Absolute positioning

It is also possible to define the position of each node manually, as a coordinate pair. The syntax is:

`\node[state] at (`⟨*x*⟩`,` ⟨*y*⟩`) (`⟨*id*⟩`) {`⟨*name*⟩`};`

The fields ⟨*x*⟩ and ⟨*y*⟩ are the $x$ and $y$ components of the position. To better illustrate this, we make use of the `help lines` option to draw a $3 \times 3$ grid, in which the nodes will be positioned.

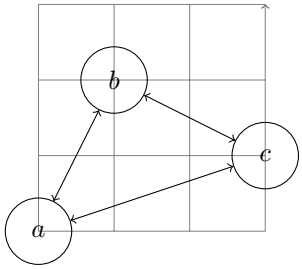


**Figure 6**: An isosceles triangle

```
\begin{tikzpicture}[<->]
    \draw[help lines] (0,0) grid (3,3);
    \node [state] at (0, 0) (a) {$a$};
    \node [state] at (1, 2) (b) {$b$};
    \node [state] at (3, 1) (c) {$c$};
    \draw
    (a) edge (b)
    (b) edge (c)
    (c) edge (a);
\end{tikzpicture}
```

### 3.2 Drawing arcs and bending edges

Especially when avoiding crossing edges in more complicated automata, it is useful to be able to draw edges as arcs (instead of straight lines). In plain Ti*k*Z it is possible to render arbitrary arcs, specifying the center, the starting and stopping angle, and the radius (as shown in [5]):

`\draw (`⟨*x*⟩`,`⟨*y*⟩`) arc (`⟨*start*⟩`:`⟨*stop*⟩`:`⟨*radius*⟩`);`

Through the automata library, however, the syntax is simplified for the case of an arc between two (already defined) nodes.

`(`⟨*id1*⟩`) edge[bend` ⟨*options*⟩`] (`⟨*id2*⟩`);`

More commonly, a directional option (`left` or `right`) is used to inform in which side the arc should be rendered. Furthermore, its radius can also be changed, by including the angle (in degrees) of the arc. Our last example illustrates these options:

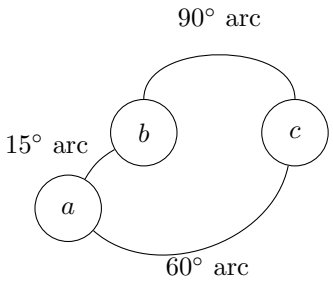


**Figure 7**: Various arcs

```
\begin{tikzpicture}
\node[state] at (0, 0) (a) {$a$};
\node[state] at (1, 1) (b) {$b$};
\node[state] at (3, 1) (c) {$c$};
\draw
(a) edge[bend left=15] node[above left]
{$15^{\circ}$ arc} (b)
(b) edge[bend left=90] node[above=0.25]
{$90^{\circ}$ arc}(c)
(c) edge[bend left=60] node[below]
{$60^{\circ}$ arc} (a);
\end{tikzpicture}
```

## References

[1] S. Sikdar. *Drawing Finite State Machines in LaTeX using TikZ: A Tutorial*, 2017.

[2] M. Sipser. *Introduction to the Theory of Computation*. Thomsom/Course Technology, 2005. Pages 48–50.

[3] T. Tantau, et al. *The TikZ and PGF Packages: Manual for version 3.1.10*, ch. Automata Drawing Library, pp. 571–575. 2023. `ctan.org/pkg/pgf`

[4] TeX Stack Exchange. Difference between "right of=" and "right=of" in PGF/TikZ. `tex.stackexchange.com/questions/9386`.

[5] TeX Stack Exchange. How is arc defined in TikZ? `tex.stackexchange.com/questions/175016`.

[6] TikZBlog. Automata diagrams in LaTeX. `latexdraw.com/automata-diagrams-in-latex`, 2021.

◇ Igor Borja
`igorpradoborja (at) gmail dot com`
`https://github.com/IgorPBorja`